Assessing the Risk of an Adaptation using Prior Compliance Verification

Allen Marshall
University of Tulsa
allen-marshall@utulsa.edu

Sharmin Jahan
University of Tulsa
shj594@utulsa.edu

Rose F. Gamble
University of Tulsa
gamble@utulsa.edu

Abstract

Autonomous systems must respond to large amounts of streaming information. They also must comply with critical properties to maintain behavior guarantees. Compliance is especially important when a system self-adapts to perform a repair, improve performance, or modify decisions. There remain significant challenges assessing the risk of adaptations that are dynamically configured at runtime with respect to critical property compliance. Assuming compliance verification was performed for the originally deployed system, the proof process holds valuable meta-data about the variables and conditions that impact reusing the proof on the adapted system. We express this meta-data as a verification workflow using Colored Petri Nets. As dynamic adaptations are configured, the Petri Nets produce alert tokens suggesting the potential proof reuse impact of an adaptation. Alert tokens hold risk values for use in a utility function to determine the least risky adaptations. We illustrate the modeling and risk assessment using a case study.

1. Introduction

As autonomous systems proliferate in all domains of use, they must become more self-adaptive to hone their decision-making performance and accuracy while accepting continuous data inputs. Self-adaptation requires a methodology to configure adaptive plans that can produce the desired changes. The MAPE-K loop (Monitor-Analyze-Plan-Execute with Knowledge) has been proffered as the standard for self-adaptation of autonomic processes [10], and therefore, has been extensively studied [2, 4, 5, 7]. There remain significant challenges in the risk assessment of adaptations that are configured dynamically at runtime during the planning portion of the MAPE-K loop with respect to the system's safety and security properties. Such properties are targeted for compliance guarantees by the deployed system and, if critical, are formally verified or certified. The difficulties of risk assessment are inversely related to the amount of pre-knowledge the autonomous system has of the adaptations it can perform. When arbitrary or unanticipated adaptations are possible at runtime, assessing the risk of compliance of the adaptations with critical properties becomes quite challenging.

If adaptive plans are configured at runtime with minimal restrictions on what changes are allowed, then they will not have been vetted during the design of the system. In this paper, we examine a risk-based evaluation strategy to compare adaptive plans configured by a self-adaptive system. The assumption is that there is little prior knowledge of the overall effect the adaptive plans will have on the system execution. The risk strategy focuses on the likelihood that the changes caused by the adaptive plan will affect the reusability of the original requirement compliance verification processes, such as formal proofs or validation and certification methods. In contrast to runtime verification, our approach abstracts and models artifacts of verification processes, such as the state variables, conditions, and dependencies that are used to prove requirements compliance prior to deployment. The premise is that the stronger the restriction on verification process reuse, the higher the likelihood that the adaptive plan will cause a requirement violation.

By transitioning the runtime reverification burden of the adaptive system to determining the risk of verification process reuse, runtime risk assessment becomes an option for a broad set of possible adaptations. In addition, when human analysis of the autonomous system is performed, the impact of adaptive plans on reverification or recertification is immediately known, along with where the impact occurred in the code.

In this paper, we focus on assessing the risk of verification process reuse given a requirement and its compliance proof. The risk is represented using probability estimates, from which we compute the expected utility of each adaptive plan with respect to verification process reuse to identify the least risky self-adaptation. Probability estimates are calculated based on verification concerns elevated from the original proof process, along with their impact on the proofs and knowledge supplied by the MAPE-K planner. We illustrate the utility calculations using a case study.

URI: http://hdl.handle.net/10125/50594
ISBN: 978-0-9981331-1-9

2. Background

Self-adaptive systems have proliferated and improved over the past decades, but verification and certification of the adaptive properties remains a challenge. Villegas, et al. [20] survey self-adaptive systems and adaptation properties derived from control theory. The framework proposed evaluates self-adaptive systems, where adaptation properties are specified explicitly and driven by quality attributes. Aspects of the framework include dimensions to classify systems, adaptation properties, mappings between adaptation properties and quality attributes, and a set of quality metrics to evaluate the adaptation properties. The framework provides an approach to measure adaptation properties and then classify them according to where and how their properties are observed.

The FeatUre-oriented Self-adaptatION (FUSION) framework proposes to measure the impact of adaptation design [8]. This approach evaluates the quality of an adaptive plan by learning the impact of adaptation decisions on the system's goals. The framework performs fine-tuning of the adaptation logic to unanticipated conditions, reducing the upfront effort required for building such systems and making the run-time analysis more efficient. FUSION learns and adapts in terms of features and can learn runtime behavior which is not considered at design time, but it does not verify that an adaptive plan complies with expected behavior.

A risk-based adaptive security framework is proposed in [1] for IoT in eHealth, which can estimate and predict risk damages and future benefits using game theory and context-awareness techniques. The framework consists of an adaptive risk management model, an adaptive monitoring model, analytics and predictive models, adaptive decision-making models, and evaluation and validation models. The adaptive risk management (ARM) model coordinates the adaptive monitoring, analytics and predictive models, adaptive decision-making models, and evaluation and validation models into a continuous cycle. ARM can learn, adapt, prevent, identify, and respond to known and unknown security threats in real-time and can develop risk-based adaptive security mechanisms. However, the demonstration of the technology appears to currently be limited to minor threats and changes.

In [19], a runtime verification framework is proposed for evaluating and validating self-adaptive behavior to guarantee compliance with security properties. The framework integrates runtime verification enablers into the feedback adaptation loop of the ASSET adaptive security framework. The framework needs four enablers: models at runtime, requirements at runtime, dynamic context monitoring, and a runtime verification. The claim is that these enablers allow for runtime verification of the outputs from feedback control loops for the validation of adaptive plans before instrumenting them, and include relevant mechanisms to keep track of the validation aspects. The resulting construct and integrated use of the enablers is not well-defined nor demonstrated.

Integrated model-based development and formal verification for self-adaptive systems could prove to be a more effective approach to verify an adaptive plan, but would still require a mechanism to compare plans should none be completely verifiable. In [18], an approach is described for an automatic slicing technique of models with respect to the properties to be verified. The modeling concepts incorporate a formal semantics and create an intermediate modeling layer on which to describe the system properties. This intermediate layer is then "sliced" to reduce the size of the models so that automatic verification can be applied. The models are then composed, retaining their verification status, to the original and more complex system models.

3. Creating Verification Workflows

Critical requirements, especially those that relate to the system architecture and expected operating environment, would be targeted for formal verification, including proof and model checking, or certification with some level of risk to ensure the safety and security of the system prior to its deployment. We use the term *verification process* to encompass the various strategies used to determine the system's compliance with critical requirements. When an autonomous system self-adapts by inserting new functionality, modifying functionality, or modifying state variables, the resulting change, called the *adaptive plan*, should fall within a risk tolerance threshold for compliance with critical requirements or provide notifications as to the extent and risk of compliance failure.

Performing runtime reverification or recertification of information system compliance in order to deploy an adaptive plan is not currently a viable solution due to the problems with those techniques [3,15]. Our approach seeks to determine the probability of reusing the original verification processes on the adapted system, such as the proof, validation, and/or certification processes, that were originally performed (and documented) to guarantee requirements' compliance. The premise is that a low probability of verification process reuse corresponds to a greater likelihood of a requirement violation. Verification process reuse can be inhibited if a state, condition, and/or control flow relied on by the original verification process has been altered by the adaptation. Thus, if multiple, comparable, adaptive

plans can be dynamically configured, the best plan to emerge would be the one with the highest probability of verification process reuse and, thus, a greater likelihood of compliance.

In this section, we formulate a *verification workflow* (VFlow) as the model of a verification process using a Colored Petri Net (CPN). The CPN provides input to a utility function (discussed in Section 4) that determines the risk of deploying an adaptive plan based on assessing the plan for verification process reuse. A VFlow is constructed for each requirement that has an associated verification process.

3.1. Verification Concerns

We derive *verification concerns* from meta-data associated with the verification process for each critical requirement. This meta-data indicates state variables, conditions, functions, methods, and components that play a role in the original verification process. Verification concerns may be derived from component interfaces (provides/requires) [9], the pre- and postconditions of Hoare triples to express safety and progress requirements [11], and state variables within test scenarios, such as those used for security control implementation and assessment [16].

As a simple example, a requirement may be expressed as a safety or progress property. One type of safety property is an invariant. To prove that properties A and B are invariant, as written in Linear Temporal Logic (LTL) [14] below,

$$\Box\ (A \wedge B)$$

it must be the case that the properties expressed by A and B hold in all reachable states. That can be restated in terms of Hoare triples, as many researchers use to express proofs and the potential for proof reuse [6].

$$[\forall s \in F \mid \{\ A \wedge B\ \}\ s\ \{\ A \wedge B\ \}]$$

means that for all statements s in program F, A and B must be preserved by the execution of s. This immediately makes A and B verification concerns. Sometimes, other state variables may affect A or B, either by being embedded into a condition that may cause them to be *false* or by having a side effect that can impact their state change. Assume that during the verification process, it is determined that

$$C \Rightarrow A$$

Then the state of C may impact the state of A. Thus, C becomes a verification concern.

Progress property proofs can follow a similar verification process to safety property proofs, except that progress properties, as temporal properties, cross states and, therefore, dependencies must be captured that exist between verification concerns. For example, the following progress property written in LTL,

$$\Box\ (D \Rightarrow \bigcirc(D \wedge K))$$

means that if D becomes *true* in some state, then in the next state (D ∧ K) will be true. This property can be stated in terms of Hoare triples, such as

$$[\forall s \in F \mid \{\ D\ \}\ s\ \{\ D \wedge K\ \}]$$

which clarifies that in a state when D holds, the next state will have D and K both *true*, since all statements must ensure this is the case.

With progress properties, the verification concerns still reflect the conditions expressed in the Hoare triples, making both D and K verification concerns. Other properties may need to hold to ensure that none of the statements that can execute inhibit K from being true or change D to false in the next state after D becomes true, establishing a dependency between the states. These properties are expressed as lemmas with their own proofs that may produce additional verification concerns. Often functions are assumed to be atomic, meaning they cannot be interrupted. Thus, a statement in F may be a function G. When proving a property, it may be necessary to examine the functions that G comprises resulting in an examination of intermediate (internal) state changes. For example, assume that $G = (g \circ f)$ and the following statements for f and g can be proven.

$$[\forall t \in f \mid \{\ D\ \}\ t\ \{\ J\ \}]$$
$$[\forall w \in g \mid \{\ J\ \}\ w\ \{\ D \wedge K\ \}]$$

Since the next complete state change occurs after G, the progress property is not violated. But if an adaptive plan did not abide by the atomicity assumptions, then the intermediate state transition could be problematic. Therefore, properties, such as J above, are also captured as verification concerns. In the case study described in Section 5, we have proven the requirements against Java code and extracted the verification concerns from those proofs, as detailed in [11].

3.2. Organizing the Verification Workflow

Each verified requirement has a related VFlow that models the verification process using the following information:

- Verification concerns as derived from the verification process.
- Where, in the system architecture, the verification concerns were of interest, which may involve multiple components and functions.

- Conditions related to the impact or prominence of verification concerns in the verification process. These conditions describe change types or ranges that fall in one of three sets related to expected impact: *devastating*, *worrisome*, or *unconcerned*.

The system architecture, as part of the VFlow construction, underlies the verification process and self-adaptive system [17]. Components are generally examined independently for compliance and, then, as part of the larger system architecture through their interfaces. Within the components are the state variables, functions, and dependencies that the verification process must also examine, in the form of lemma proofs [11]. Thus, the VFlow should represent the proper granularity of the architecture description that best fits the verification process perspective and flow.

3.3. Constructing the Colored Petri Net

A Petri Net is a bipartite graph that makes available mathematical analyses and decision processes to a wide range of applications by allowing the expression of system functionalities and architectures. Colored Petri Nets (CPNs) introduce additional functionality through distinguishing features among tokens traversing the network and complex processing by the transitions that dictate token paths. CPNs have been successfully applied to modeling scenarios in such areas as distributed decision making, attack modeling, access control policies, web service composition, process control, and the software development process [12, 13]. Reachability, conflict detection, deadlock detection, and quality of service are a few of the major properties that CPNs can assess. Their flexibility in modeling architectures and component processing, along with the availability of automated tools to simulate them, make them an appropriate representation for VFlows used to assess the risk of an adaptive plan.

The current representation of a VFlow models the major components important to the verification processes as places in the CPN. A generic VFlow is shown in Figure 1. Tokens traverse the places using transitions. In a CPN, transitions can perform complex processing based on embedded, immutable data structures.

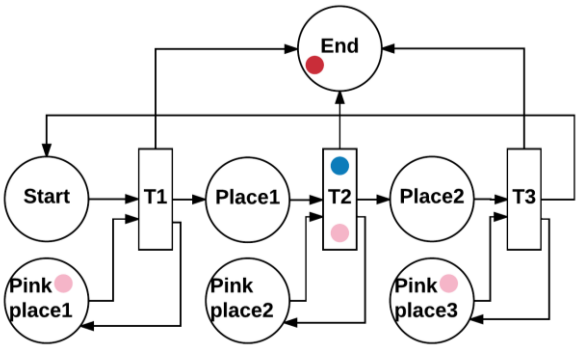

Figure 1. Generic VFlow represented as a CPN

There are three token colors used in the VFlow. Pink tokens hold the adaptive plan's *change set*. The planning portion of the MAPE-K loop formulates and configures potential plans to be risk-assessed. The assumption is that the change sets embody information related to (1) what parameters are changed, (2) how, in general, they are changed, (3) what major components are affected, and (4) what the planner believes the impact of that change to be overall. The *change impact* can be determined by the planner based on accumulated knowledge through techniques such as reasoning over the success or failure history of changes, machine learning outcomes based on adaptive plans shared across related information systems, and partial plan simulations. More details on the calculation of the change impact are given in Sections 4 and 5. The structure of the pink token appears below where each element of the change set has a unique ID, the verification concern affected (VC), the type of effect to the verification concern (condition), and its planner-calculated change impact ($\hat{p}$).

$$t_{pink} = ((ID_1,\ VC_1,\ condition_1, \hat{p}_1),\ ...,\ (ID_n,\ VC_n,\ condition_n, \hat{p}_n))$$

The blue token traverses the CPN looking for verification concerns in the verification process that may be affected by the change set. These are called *conflicts*. This token holds the outcomes from affected verification concerns as well as change sets provided by pink tokens. Thus, checks against verification concerns can be performed at all places in the CPN, regardless of where the pink token designates the change will occur. As seen below, the blue token tracks the places traversed (visited) so that it ensures that every verification concern in the change set is examined at every place. The tracking also determines when the blue token's cycle is complete and it can be absorbed to terminate the CPN.

$$t_{blue} = (\text{visited, vcMatches, vcConflicts, dependencies, conflictCount, tokenCount})$$

The set vcMatches in the blue token accumulates the conflicts found as a set of tuples of the form ($ID_{conflict}$, vcInfo, conflictPlace, $\hat{p}$). $ID_{conflict}$ is a unique identifier of the conflict based on the blue token's conflictCount. This identifier is used for the alert and may be repeated across alerts if more than one place is affected. vcInfo holds a record of the verification concern affected and the impact determined by the transition based on the change set's condition for that verification concern. Thus, vcInfo embodies the VFlow's impact indicator which may be different from the change set's impact indicator, $\hat{p}$. The conflictPlace is where the conflict was found. Each conflict in vcMatches will be at a unique place because a verification concern can only appear once at a place. If any change to a verification concern can strongly impact the risk of reusing the verification process for the requirement, it will be reflected in the impact indicator in vcInfo.

The set vcConflicts in the blue token holds the pink token's information for comparison with information at each place in the VFlow as the blue token traverses the CPN. As discussed in Section 3.1, progress properties embody dependencies among state variables. These are captured in the blue token's dependencies set, which allows the blue token to manage the dependencies by enforcing a check on the impact of a verification concern at a place before or after a conflict was already found with its dependent verification concern. The conflictCount generates the unique ID for each pink token information, while the tokenCount generates a unique ID per red token.

Red tokens are output by transitions to represent alerts. These alerts indicate potential conflicts between the adaptation's change set and the requirement's original verification process. The structure of a red token is as follows.

$$t_{red} = (\text{ID}, ID_{conflict}, \text{vcInfo}, \hat{p}, \text{conflictPlace}, \text{placeStatus})$$

The red token ID and $ID_{conflict}$ are assigned by the blue token prior to the red token being sent to the end state. The red token must hold all of the factors needed for the risk assessment of the adaptive plan for that VFlow. The set vcInfo contains tuples of the form (VC, vcImpact), so that the transitions' impact factor based on the pink token's change set condition is recorded. The pink token's $\hat{p}$ value is also maintained in the red token along with the place where the conflict occurred and the weight of that place's importance to the verification process.

An example transition appears as pseudo code below. This is one of 18 transition rules used in a VFlow based on our modeling of verification processes for safety and progress properties. Transitions always require a blue token and a pink token as input. In this example, the blue token, B, has not visited the input place to the transition. The transition, T, does not have a verification concern (VC) that conflicts with what the blue token has accumulated in B.vcConflicts. The transition does have a conflict with a VC in the pink token P's change set.

Transition Conditions:
T.place_name is not in B.visited
No VC in T.vcInfo appears in B.vcConflicts
A VC in P conflicts with a VC in T.vcInfo

Transition Actions:
FORALL VC in T.vcInfo that appear in P
Increase B.count and B.$ID_{conflict}$
Create a red token, R, with B.count as its ID, B.$ID_{conflict}$ as its $ID_{conflict}$, and other information held by P and T
Update B.vcMatches to include $ID_{conflict}$ and the appropriate information held by T for all matching VCs
Add the appropriate change information from P to B.vcConflicts using B.count for $ID_{conflict}$
FORALL VC in P that are not in T.vcInfo
Increase B.count and B. $ID_{conflict}$
Add the appropriate change information from P to B.vcConflicts using B.count for $ID_{conflict}$
Add T.placeName to B.visited
Send B to output place
Send P to its place of origin
Send all red tokens to end place

4. Evaluating Risk

For each requirement r and adaptation plan a, the VFlow outputs a set of red tokens $T(r, a)$ representing alerts. Once the red tokens are generated, the system calculates a metric that can be used to compare the risks of adaptation plans based on the token information. To obtain a workable formula, we assume that each red token, independently of all other tokens, has the potential to represent an *actual violation* of verification process reuse. That is, the adaptive plan has altered something that was relied on by the original proof of the requirement. We also assume that verification process reuse is violated if and only if there is at least one red token representing an actual impact. (Our current work ignores the possibility that a group of red tokens might represent the violation in combination, without doing so individually.) For each red token t, let $S(t)$ be an indicator variable with value 0 if t represents an actual violation and 1 otherwise. For each requirement r and adaptation plan a, let $I(r, a)$ be an indicator with value 0 if a violates the reuse of r's proof and 1 otherwise. Although the values of $S(t)$ and $I(r, a)$ would typically be deterministic, we assume they are infeasible to compute, and therefore model $S(t)$ and $I(r, a)$ as

random variables. Our assumptions given above translate to the following statements.

For each requirement r and plan a,

$$I(r,a) = 0 \Leftrightarrow \big(\exists\, t \in T(r,a)\big)\big(S(t) = 0\big).$$

For each requirement r and plan a, the random variables in the set

$$\bigcup_{t \in T(r,a)} \{S(t)\}$$

are mutually independent.

From these statements, we deduce that the probability that plan a does *not* violate the reuse of requirement r's proof is

$$P(I(r,a) = 1) = \prod_{t \in T(r,a)} P(S(t) = 1).$$

To compare adaptation plans, we define the *requirement utility* of plan a to be the weighted sum

$$U(a) = \sum_{r \in R} w(r) I(r,a)$$

where R is the set of requirements and $w(r)$ is a stakeholder-supplied *utility weight* for the need to maintain system compliance with requirement r. The expected requirement utility is then as follows.

$$E[U(a)] = \sum_{r \in R} \left(w(r) \prod_{t \in T(r,a)} P(S(t) = 1) \right)$$

If the expected requirement utility can be computed, it can be used as a metric to distinguish riskier plans from less risky plans. However, this goal requires an estimate $p(t) \approx P(S(t) = 1)$ for each token t. Each red token contains such an estimate $\hat{p}(t)$, based on whatever knowledge the planner may have to compute it. Since $\hat{p}(t)$ is presumed to have been computed without consideration of the characteristics of the original verification process, we wish to adjust it based on proof-related information to get the final estimate $p(t)$.

Our approach to computing $p(t)$ is based on the idea that verification concerns and architectural places can have differing levels of prominence or impact in a verification process. We assume that red tokens generated from a high-impact place or verification concern are more likely to represent actual reuse violations than those coming from low-impact places or concerns. A red token t contains impact multipliers $M_{PL}(t)$ and $M_{VC}(t)$, representing the impact of the architecture place and verification concern (respectively) from which the token was generated. Lower multipliers represent higher impact/risk. We apply these multipliers to $\hat{p}(t)$ to get $p(t)$, resulting in an estimate that takes into account the proof characteristics.

We have considered two possible ways to apply the multipliers: scaling the probability and scaling the odds. The latter allows for the possibility of multipliers greater than 1, which would indicate that the estimate given by the planner should be increased rather than decreased. One of the outcomes of this study is a comparison of the two approaches based on how well they estimate the relative risk of adaptations.

With probability scaling, we have

$$p(t) = M_{PL}(t) M_{VC}(t) \hat{p}(t).$$

With odds scaling, we have

$$p(t) = \frac{o(t)}{1+o(t)}, \text{ where } o(t) = \frac{M_{PL}(t) M_{VC}(t) \hat{p}(t)}{1-\hat{p}(t)}.$$

(If $\hat{p}(t) = 1$, this formula is undefined, and we instead use $p(t) = 1$.)

5. Case Study

To evaluate our risk comparison methodology, we apply it to a case study that we have developed previously, called the Multi-Mode Traveler System [11]. The case study involves a system on which we impose multiple self-adaptive plans. As the system is quite simple, we can manually reason about and compare the risks of each adaptation. Our goal is to determine whether the more mechanistic utility comparison metric described in Section 4 can come to conclusions similar to those which we have derived manually, given the original verification processes.

The MMTS consists of a traveler that moves in a grid while attempting to avoid stationary enemies distributed randomly on the grid. At each step, the traveler attempts to choose a new position and move to it. Based on the direction of the move, the traveler's fuel level may increase, decrease, or stay the same when it reaches the next position. The traveler is given an upper and lower limit on its fuel value, and must keep the fuel within that threshold. More complex variants of the system employ mission planning and enemy avoidance.

The MMTS base code provides an update process that chooses and sets the new position and fuel value. We identified three high-level architectural components that comprise the update process. The first is

getCurrentStatus (gCS), which reads and validates the state at the start of the update. The second is getNextPosition (gNP), which determines the set of valid moves and randomly chooses a move from that set. The third is setPosition (sP), which moves the traveler to the chosen position and updates its fuel level. These three components form the architectural places in the VFlow for the two of the MMTS requirements, R1 and R2, stated in LTL below.

R1: $\Box$ (minFuel $\leq$ fuel $\leq$ maxFuel)
R2: $\Box$ ((canMove $\wedge$ position = p) $\Rightarrow$ $\bigcirc$notAt(p))

For R1, the fuel level must stay within the threshold at all times. For R2, if the traveler can move at a given time step, then it must move. canMove is defined as validMoves $\neq \emptyset$, where validMoves is computed according to the current traveler state to exclude positions containing enemies and moves that would lead to a fuel threshold violation. A random move from the validMoves set is chosen in each update. If the set is empty, the traveler stays in its current position. notAt(p) is defined as position $\neq$ p, given p represents the current position. For the current case study, we assume the stakeholder-supplied utility weights of the requirements are $w(\mathrm{R1}) = 0.75$ and $w(\mathrm{R2}) = 1$.

In [11], we verified that the MMTS code (without adaptation) satisfies these requirements and derived the verification concerns using the process outlined in Section 3.1. As an outcome of the verification process, we extract impact multipliers for use in the risk comparison calculations. These multipliers are based on the prominence of different verification concerns and architectural places in requirements' proofs, as well as the conditions required by the proofs.

Table 1 shows the place impact multiplier for each of the 3 architectural places for each requirement. In this case study, we manually assigned values of 0.2 (high impact), 0.5 (medium impact), or 0.9 (low impact) based on our perception of the importance of each place in each proof. These values and others described in this section are shown in the example VFlow for R1 in Figure 2.

Table 1. Impact multiplier for the VFlow place if a change occurs

	gCS	gNP	sP
R1	0.9	0.5	0.2
R2	0.5	0.5	0.2

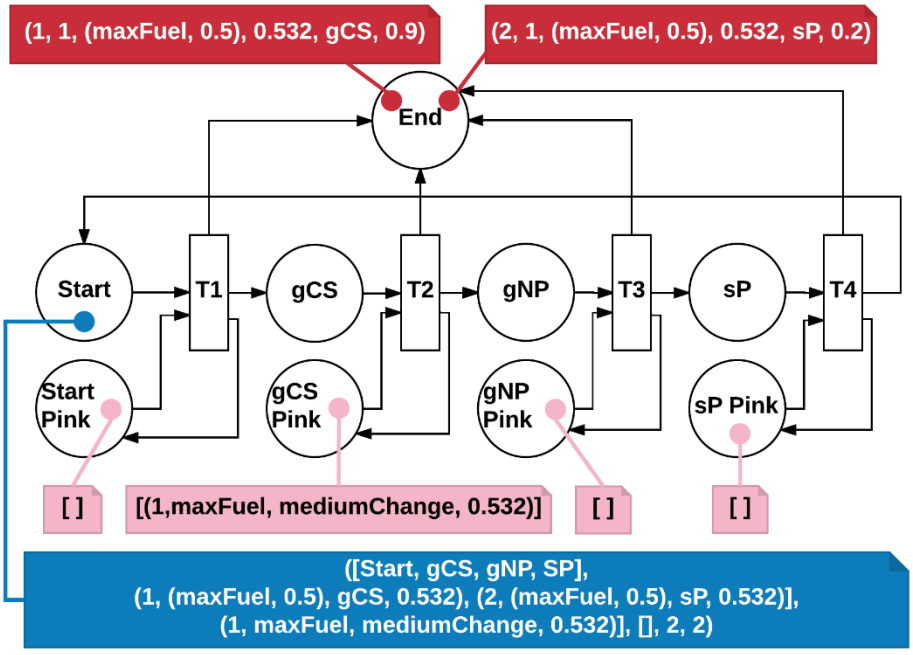

Figure 2. VFlow for R1 with the change set from adaptation A1

Table 2. Verification concern impacts from the VFlow perspective for R1

R1	devastating	worrisome	unconcerned
fuel	Change greater than or equal to $maxFuel - minFuel$, positive or negative.	Change greater than or equal to $\frac{maxFuel-minFuel}{2}$, positive or negative.	Change less than $\frac{maxFuel-minFuel}{2}$, positive or negative
minFuel	Change greater than or equal to $maxFuel - minFuel$, positive.	Change greater than or equal to $\frac{maxFuel-minFuel}{2}$, positive.	Negative change or change less than $\frac{maxFuel-minFuel}{2}$.
maxFuel	Change greater than or equal to $maxFuel - minFuel$, negative.	Change greater than or equal to $\frac{maxFuel-minFuel}{2}$, negative.	Positive change or change less than $\frac{maxFuel-minFuel}{2}$.

Table 3. Verification concern impacts from the VFlow perspective for R2

R2	devastating	worrisome	unconcerned
fuel	Set to 0.	Large change, positive or negative.	Small change, positive or negative.
minFuel	Set to maxFuel.	Increased to a value less than maxFuel.	Set to 0.
maxFuel	Set to 0.	Decreased to a value greater than minFuel.	Increased.
validMoves	Changed to a nonempty set with no change in nextMove.	Changed to a nonempty set with a change in nextMove.	Set to ∅.
nextMove	Set to null with no change in validMoves.	Set to null with a change in validMoves.	Set to null with validMoves changed to ∅.
position	Changed with nextMove set to null.	Maintained with nextMove set to null.	Maintained when validMoves is empty.

As discussed previously, the impact multiplier for a verification concern can depend on the type of change made. Verification concern impacts are categorized as *devastating*, *worrisome*, or *unconcerned*, and the corresponding impact multiplier values are 0.2, 0.5, and 0.9, respectively. Table 2 and Table 3 show the rules that we used for categorizing the relevant verification concerns' impacts for R1 and R2, respectively. The categorization is based on the type and/or magnitude of the change, which is supplied by the planner.

5.1. Potential Adaptations

In our example scenario, the MMTS is initialized with the traveler at position (0,0) with a fuel value of 80, a lower fuel threshold minFuel = 0, and an upper fuel threshold maxFuel = 160. The system is simulated for 25 time steps, at which point an engine failure occurs, triggering the adaptation process. The planner configures possible adaptation plans and constructs their change sets to assess the risk of impacting proof reuse for requirements R1 and R2.

The four adaptation plans we compare are:

A1: maxFuel is divided by 2 within gCS.

A2: maxFuel is reduced by 40, and minFuel is increased by 40 within gCS.

A3: The next move for the traveler is not chosen for 5 time steps within gNP, even if validMoves is nonempty (to simulate a stop for repair).

A4: validMoves is changed to the empty set for 5 time steps (to simulate a stop for repair) in gNP.

5.2. Examining Adaptation Plan Risks

By manual analysis and simulation, we have identified the potential risks of each plan with respect to the requirements. A1 is risky for R1, as it leads to a violation if the current fuel value is greater than 80. When R1 fails, it also causes a failure in R2, because the traveler stops moving if it detects a violation of the fuel threshold. Therefore, A1 is also risky for R2. A2 can also pose a threat to R1 and R2 if the current fuel value is below 40 or above 120, but that is not possible given the function for calculating the fuel by the 25th time step when the adaptation occurs. Therefore, A2 has very little risk if performed early in the traveler movement.

A3 and A4 both disallow movement for 5 moves, which poses no threat to R1. A3 is very risky for R2, and in fact will always cause a violation if the set of valid moves is nonempty at the time of the adaptation. A4 is superficially similar to A3, but it actually is not risky for R2, because R2 only requires movement when the set validMoves is nonempty.

Based on our analysis, A3 should be considered risky for R2, while A1 should be considered risky for R1 and R2. A2 might be considered marginally risky for R1 and R2, because it could fail under some circumstances, although those circumstances are not possible in our simulation. A4 poses no risk for either of the requirements.

It remains to be shown how the planner generates the initial success probability estimate $\hat{p}$ for each plan's pink token(s). In this case study, we assume the planner knows that reducing the set of valid moves is less risky, so it uses $\hat{p} = 0.99$ for all of A4's pink tokens. We assume the planner has been able to determine that changing the logic in getNextPosition has some risk to both requirements, so it uses $\hat{p} = 0.5$ for A3's pink tokens.

For A1 and A2, we assumed that the planner might run predictive simulations involving perturbations of minFuel or maxFuel, to estimate the sensitivity of the system to such changes. We performed 200-step simulations in which either minFuel or maxFuel was perturbed at the 100th step, with 1000 runs of the simulation for each perturbation size and requirement. The simulations produced a success rate of 0.532 for R1 when maxFuel was reduced by 80, meaning 53.2% of the 1000 simulations found no violation of R1. The corresponding success rate for R2 was 0.505. Therefore, we set $\hat{p} = 0.532$ for A1's pink token in R1's VFlow, and $\hat{p} = 0.505$ for the pink token in R2's VFlow.

In all simulations where minFuel was increased by 40 or maxFuel was decreased by 40, no proof violations were detected. Assuming the planner would be cautious enough not to indicate a guaranteed success based on a simulation, we set $\hat{p} = 0.99$ for all of A2's pink tokens.

5.3. Computed Results

Table 4 and Table 5 show the success probabilities computed from the set of red tokens for each requirement/plan pair, along with the expected utility based on the probabilities and the requirements' utility weights. Table 4 provides the results for the probability scaling approach. Table 5 shows the odds scaling results.

Table 4. Results from probability scaling approach

Results Using Probability Scaling			
	R1 Success Prob.	**R2 Success Prob.**	**Expected Utility**
A1	0.0127	0.00708	0.0166
A2	0.0204	0.00630	0.0216
A3	1	0.0500	0.800
A4	1	0.446	1.20

Table 5. Results from odds scaling approach

Results Using Odds Scaling			
	R1 Success Probability	**R2 Success Probability**	**Expected Utility**
A1	0.0345	0.0226	0.0485
A2	0.875	0.858	1.51
A3	1	0.0909	0.841
A4	1	0.978	1.73

6. Discussion and Conclusion

The risk values calculated using odds scaling match fairly well with what we would expect based on our manual reasoning. For R1, A3 and A4 were found to have no risk, A2 was found to have low risk (high success probability), and A1 was found to have high risk. For R2, A1 and A3 were found to have high risk, while A2 and A4 had low risk. However, there are some discrepancies from what we would expect. For example, the success probability computed for R2 was higher for A3 than for A1, even though A3 nearly always causes a failure for R2 while A1 causes failures less frequently. The same issue occurs when probability scaling is used. An additional problem occurs with probability scaling in that A2 is found to have the lowest success probability for R2, even though it is one of the safer adaptations for that requirement.

Given these discrepancies, we plan to refine our risk calculations in future work. In particular, we will attempt to find a way to relax the assumption that all red tokens affect the requirements independently of each other. Another important future task will be to formalize the process of extracting meta-data from a verification process. Our approach thus far has relied on manual proofs and manual meta-data extraction. We plan to develop a methodology for extracting this information in a way that can be automated in some form.

7. Acknowledgment

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-16-1-0248. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

8. References

[1] Abie, H. and I. Balasingham, "Risk-Based Adaptive Security for Smart IoT in eHealth", Proceedings of the 7th International Conference on Body Area Networks, ICST, Oslo, Norway, 2012.

[2] Almeida, R. and M. Vieira, "Benchmarking the Resilience of Self-Adaptive Software Systems: Perspectives and Challenges", Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, ACM, Waikiki, Honolulu, HI, USA, 2011.

[3] Calinescu, R., C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-Adaptive Software Needs Quantitative Verification at Runtime", Communications of the ACM, ACM, New York, USA, Vol. 55, No. 9, pp. 69-77, 2012.

[4] Camara, J., R. de Lemos, C. Ghezzi, and A. Lopes, (Eds.), Assurances for Self-Adaptive Systems: Principles, Models, and Techniques, vol. 7740, Springer-Verlag, 2013.

[5] Cheng, B., R. de Lemos, H. Giese, P. Inverardi, and J. Magee, (Eds.), Software Engineering for Self-Adaptive Systems, vol. 5525. Springer-Verlag, 2009.

[6] Damiani, F., J. Dovland, E. Johnsen, and I. Schaefer, "Verifying Traits: A Proof System for Fine-Grained Reuse", Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs, ACM, Lancaster, UK, 2011.

[7] de Lemos, R., H. Giese, H. A. Müller, and M. Shaw, (Eds.), Software Engineering for Self-Adaptive Systems II, vol. 7475. Springer-Verlag, 2013.

[8] Elkhodary, A., N. Esfahani, and S. Malek, "FUSION: a Framework for Engineering Self-Tuning Self-Adaptive Software Systems", Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, Santa Fe, USA, 2010.

[9] Georgiadis, I., J. Magee, and J. Kramer, "Self-Organizing Software Architectures for Distributed Systems", 1st Workshop on Self-Healing Systems, ACM, Charleston, USA, 2002.

[10] IBM, "An Architectural Blueprint for Autonomic Computing", Autonomic Computing White Paper, 3rd Edition, IBM, USA, June 2005, available at http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf.

[11] Jahan, S., A. Marshall, and R. Gamble, "Embedding Verification Concerns in Self-Adaptive System Code", 11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, IEEE, Tucson, AZ, USA, 2017.

[12] Jensen, K. and L.M. Kristensen, "Colored Petri Nets: A Graphical Language for Formal Modeling and Validation of Concurrent Systems", ACM, Communications of the ACM, New York, 2015, Vol. 58, No. 6, pp. 61-70.

[13] Jensen, K. and L.M. Kristensen, Coloured Petri Nets: Modelling and Validation of Concurrent Systems, Springer-Verlag, 2009.

[14] Lichtenstein, O. and A. Pnueli, "Checking that Finite State Concurrent Programs Satisfy their Linear Specification", Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, New Orleans, USA, 1985.

[15] Newcombe, C., "Why Amazon Chose TLA+", Abstract State Machines: Alloy, B, TLA, VDM, and Z, LNCS 8477, Y. Ameur and K. Schewe (Eds.), Springer, Toulouse, France, 2014.

[16] NIST, Assessing Security and Privacy Controls in Federal Information Systems and Organizations, Special Publication 800-53 Revision 4, NIST, April 2013.

[17] Oreizy, P., N. Medvidovic, and R.N. Taylor, "Runtime Software Adaptation: Framework, Approaches, and Styles", International Conference on Software Engineering, ACM, Leipzig, Germany, pp. 899-999, 2008.

[18] Schaefer, I. and A. Poetzsch-Heffter, "Slicing for Model Reduction in Adaptive Embedded Systems Development", Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, ACM, Leipzig, Germany, 2008.

[19] Torjusen, A.B., H. Abie, E. Paintsil, D. Trcek, and Å. Skomedal, "Towards Run-Time Verification of Adaptive Security for IoT in eHealth", Proceedings of the 2014 European Conference on Software Architecture Workshops, ACM, Vienna, Austria, 2014.

[20] Villegas, N.M., H.A. Müller, G. Tamura, Laurence D., and R. Casallas, "A Framework for Evaluating Quality-Driven Self-Adaptive Software Systems", Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, ACM, Waikiki, Honolulu, HI, USA, 2011.